Differentiated Scheduling of Response-Critical and Best-Effort Wide-Area Data Transfers

Rajkumar Kettimuthu,* Gagan Agrawal,† P. Sadayappan,† Ian Foster*
* Argonne National Laboratory
E-mail: {kettimut, foster}@anl.gov
† The Ohio State University
E-mail: {agrawal, saday}@cse.ohio-state.edu

***Abstract*—Many science applications that use wide area networks are *response-critical*, meaning that they need data to be delivered by a deadline. Yet the state of the art in science networks is *best-effort*, i.e., transfers are scheduled as they are submitted, with no assurance of completion time or transfer rate. Building on the observation that both the start time and concurrency associated with a given transfer can be controlled, we formulate a bi-objective file transfer scheduling problem. With *value functions* used to capture the importance and urgency of response-critical transfers, we aim to (a) maximize the aggregate value provided to response-critical transfers, while (b) minimizing average slowdown for other transfers. We present an algorithm, `RESEAL`, that provides differentiated service to transfers with timing constraints by controlling the scheduled load at the transfer endpoints, while also minimizing the impact of those transfers on other (best-effort) transfers by delaying time-constrained transfers, where useful, so that they complete as close as possible to their optimal completion times (time after which their value starts to decrease). We evaluate `RESEAL` in a production wide-area network environment using real-world transfer logs. We show that the algorithm can allow response-critical transfers to achieve an aggregate value of 90% of their maximum aggregate value, even when the total load on the network is as high as 60%, with only 9% slowdown for best-effort tasks. Our results suggest that the needs of response-critical applications can be met without resource reservations.**

I. Introduction

Data movement is an essential component of distributed science [7], [23]. As data volumes increase, traditionally single-institutional science workflows increasingly use remote computational resources for analysis, visualization, and archival purposes [8], [14]. Thus, workflow performance becomes dependent on wide-area data transfer efficiency. The current state of the art in data movements can be described as *best-effort*—each transfer is scheduled as it is requested, without considerations of its impact on other transfers and without any differentiation between different transfer types [4].

Our focus in this paper is on use cases in which data transfers have timing constraints. One example is the remote analysis of experimental data that can guide the selection of parameters for the next experiment or even influence the course of the current experiment. Another example is the transfer of input data for a computation that has already acquired computational resources. Currently available tools for data movement in the high-performance computing (HPC) environment, such as the Globus transfer service [4], GridFTP [2], BBCP [6], and HPN-SCP [24], have not been designed to provide the differentiated service required by such transfers, although other forms of optimizations on data transfers have been applied [15], [11]. More broadly, differentiated mechanisms for scheduling file transfers have received only little attention [18].

One approach to catering to the requirements of response-critical transfers is resource reservation. A number of projects support the provisioning of dedicated network channels [22], [40], [35], [42]. Advanced networks such as ESnet, Internet2, and GEANT provide the capability to create virtual circuits and reserve bandwidth on the wide-area network. This capability has been used to improve the performance of wide-area data transfers in certain environments [39], [31]. Foster et al. [18] used advance reservation and co-reservation of heterogeneous resources for end-to-end quality of service. StorNet [21] uses co-reservation of storage and network resources to provide guaranteed disk-to-disk data transfer performance.

We describe here a different, application-level approach to providing differentiated services for file transfers, based on controlling when transfers are initiated and the degree of concurrency allocated to each transfer. We classify transfer requests as *best-effort* (*BE*) or *response-critical* (*RC*). A *BE* task needs only to be transferred as soon as possible, whereas an *RC* task has time constraints. We use a model to estimate the external load on the link over which a transfer is to be performed, and we adjust the concurrency level to control the throughput achieved for each transfer [28]. For RC tasks, we use a *value function* to capture both the maximum value if the task is completed within a specified window and its decay if the task is delayed beyond that window.

With this background, we formulate a bi-objective scheduling problem, where the goal is to both maximize the aggregate value for *RC* transfers and minimize the average slowdown for *BE* transfers. Building on the `SEAL` (*SchEduler Aware of Load*) load-aware transfer scheduling algorithm developed in previous work [29], we develop a new algorithm for this bi-objective problem. `SEAL` minimizes the average slowdown across all transfers by monitoring external load and then controlling scheduled load so as to maximize the utilization of available resources without overloading them. The new algorithm, *Response-critical Enabled SEAL*: `RESEAL`, accommodates *RC* tasks in addition to *BE* tasks. It provides appropriate levels of service to *RC* tasks by controlling concurrency levels at file transfer sources and destinations. Moreover, it minimizes slowdown for *BE* tasks by scheduling them ahead of those *RC* tasks that can be delayed without losing their value.

We experimentally evaluate `RESEAL` in a production wide-area network (WAN) environment using real transfer logs under different load conditions. We show that `RESEAL` can achieve 96.2%, 87.3%, and 90.1% of the maximum aggregate value for *RC* tasks for transfer logs with loads 25%, 45%, and 60%, respectively, with only 2.6%, 9.8% and 8.9% increase in slowdown for *BE* tasks. We also find that load variation in the logs impacts performance significantly. For example, in a log with 45% average load but relatively higher variation in load over time, the achieved value for *RC* tasks is 87.3% of the maximum aggregate and the relative slowdown of *BE* tasks is 9.8%. These two values improve to 92.7% and 5.8%, respectively, in another log where the average load is still 45% but the variation in load over time is lower.

II. Motivation

We provide more background on the need for response-critical wide-area data transfer and discuss potential approaches to meeting these needs.

A. Motivating Science Cases

The need to ensure timely completion of data transfers arises in multiple science communities [23], [1], [7], [8]. Many relevant science cases involve an instrument that produces data: as data volumes (and thus computational requirements) increase at light sources, fusion reactors, and other experimental facilities, sufficient computing power is no longer available locally. Thus, researchers need to depend on remote computing facilities. Data transfer and analysis must happen in a timely manner in order to check results, adjust the experimental setup, and maximize the use of experimental facilities.

Another use case arises because of the emergence of clouds. For example, government agencies are funding on-demand compute resources (e.g., Magellan [37], Jetstream [27], and Bridges [9]) to meet the emerging response-critical analysis requirements. But we need mechanisms to move data between experimental facilities and compute facilities in a scheduled manner to use such resources for data-intensive applications. As a specific example, scientists from Pacific Northwest National Laboratory (PNNL) use x-ray tomography at the Advanced Photon Source at Argonne National Laboratory (ANL) for high resolution imaging. Each experiment generates several gigabytes of data. They need to process each new sample rapidly in order to make decisions that affect subsequent samples. Such analysis can be done on the on-demand computational environment at PNNL. However, data must first be moved from ANL to PNNL within a fixed time, and then results moved back to ANL.

B. Resource Reservations

Some studies have used network reservation to achieve high throughput for wide-area transfers in certain scenarios [39], [31]. We argue that this approach is neither effective nor efficient. To the former, wide-area network is only one of the many resources involved in the end-to-end file transfer; local network at the facilities, data transfer node, storage area network, and storage system resources are all shared resources involved in the data transfer. In other words, reserving only the wide-area bandwidth is not sufficient to address the requirements of response-critical file transfer use cases in general. This situation has been validated in a study by Liu et al. [36]; more specifically, it is claimed that CPU resources on the data transfer nodes must be reserved and contention for disk I/O must be eliminated. Other studies have used co-reservation of network and storage resources to achieve quality of service for wide-area data transfers [21]. In the current HPC environment, most common deployments have data transfer nodes and compute nodes mount a global shared parallel storage system, which is connected through a storage area network [13], [30], [36]. In such scenarios, one cannot ensure contention-free access to the storage system.

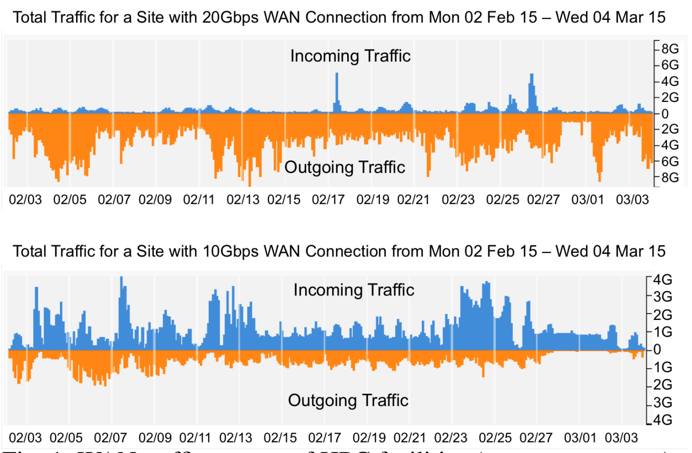


Fig. 1: WAN traffic pattern of HPC facilities (source: my.es.net)

C. Expoiting Overprovisioning

We now show why reservations may not be necessary for ensuring response-critical transfers. Backbone networks are typically overprovisioned in order to sustain peak loads; thus, they are underutilized most of the time. For example, Internet2 has a policy of operating their network at light loads (25–30%) to allow the network to absorb surges in traffic [25]. Internet2 has a published backbone upgrade practice of upgrading the backbone interconnects when the weekly 95th-percentile metric is reliably above 30% of link capacity. Fig. 1 shows the wide-area traffic for a one-month period for two HPC facilities with a 20 Gbps and a 10 Gbps wide-area connection, respectively. Although the peak rates are as high as 60%, the average is lower than 30%.

The difficulties of reserving resources and this pattern of overprovisioning motivate us to pursue an alternative approach to meeting *RC* task needs. Specifically, we develop file transfer scheduling heuristics that provide appropriate levels of service to *RC* tasks by controlling the concurrency levels associated with different transfers. We have shown in previous work that the allocation of bandwidth to different transfers can be controlled by varying their concurrency [28]. Here, we leverage this method to differentially schedule transfers with different service requirements. This control can be implemented from the file transfer application at transfer endpoints, and thus it requires no new features in network devices.

III. Problem definition and metrics

We first present prior work on load-aware scheduling and then describe the problem that we address here.

A. Precursor Work: SEAL Algorithm

In recent work [29], we developed a load-aware file transfer scheduling algorithm (*SchEduler Aware of Load*: `SEAL`) that queues, preempts, and dynamically adjusts transfer concurrency to reduce the average slowdown of file transfer tasks. Slowdown refers to the factor by which a file transfer is slowed relative to the time it would take on an unloaded system. *Slowdown* (or *bounded slowdown*), a metric commonly used in parallel job scheduling [17], is defined as:

$$BS = \frac{\text{Waittime} + \max\left(\text{Runtime}, \text{bound}\right)}{\max\left(\text{Runtime}, \text{bound}\right)}. \quad (1)$$

where *bound* is a user-defined threshold used to limit the influence of extremely short jobs on slowdown. `SEAL` defines a variant of it to suit the data transfer context (see below) and uses it as the optimization metric,

$$BS_{FT} = \frac{\text{Waittime} + \max(\text{Runtime}, \text{bound})}{\max(TT_{ideal}, \text{bound})}, \quad (2)$$

where TT_{ideal} is the estimated transfer time (TT) under zero load and ideal concurrency. TT_{ideal} is computed by using a model developed in previous work [28]. In the rest of this paper, we refer to BS_{FT} simply as *slowdown*.

SEAL aims to keep the number of concurrent transfers just enough to saturate the system. It queues transfers under high-load conditions and increases the concurrency levels of individual transfers at low-load conditions. It combines the observed performance of current transfers and transfer throughputs estimated with models developed in previous work [28] to identify system saturation and appropriate concurrency levels. If the system resources are saturated, SEAL might preempt one or more active transfers and schedule waiting transfer requests, in order to reduce the overall average slowdown of the transfer requests. SEAL has been shown to improve the average slowdown of transfers by up to 25%.

In this paper, we extend SEAL (which treats all transfer requests as best-effort) to consider two types of transfer requests: *Response-critical* (RC) and *best-effort* (BE). *RC* transfer requests have timing constraints, whereas *BE* transfer requests seek best-effort service.

B. Incorporating RC Tasks

Since *RC* tasks have timing constraints, slowdown by itself is not a suitable metric for these tasks. Instead, and as has been done in computation scheduling, we allow users to associate a *utility* or *value function* with each task. This function gives the value or utility of a task as a function of the task's slowdown. Typically, a task's value will start high but then decline over time if the task is delayed. A value function thus captures a task's importance and urgency.

Fig. 2 shows an example value function with linear decay, similar to those used in compute scheduling [26], [12], [41]. Since science data transfer tasks do not currently have value functions defined, we define one for each *RC* task in our logs as follows:

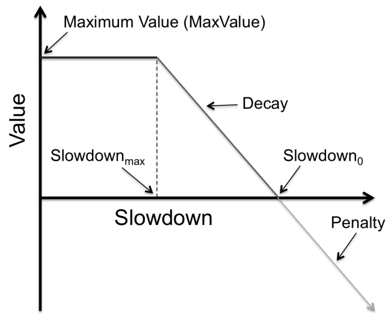


Fig. 2: Example value function

$$\text{Value} = \begin{cases} \text{MaxValue if } Slowdown \leq Slowdown_{max} \\ \frac{\text{MaxValue} \times (Slowdown_0 - Slowdown)}{(Slowdown_0 - Slowdown_{max})} \text{ otherwise} \end{cases} \quad (3)$$

where

$$\text{MaxValue} = A + log(\text{size}), \quad (4)$$

with *size* being the size of the file in gigabytes and A a constant to avoid small jobs being completely unattractive to the system.

Specifically, each *RC* task has a maximum value (*MaxValue*) that is obtained when the task completes with a slowdown less than or equal to a specified limit (*Slowdown*$_{max}$). If the task's slowdown goes above the specified limit, the value decays linearly (*Slowdown*$_0$ is the slowdown at which the value becomes zero). Hence, we have a bi-objective problem of maximizing the aggregate value obtained for *RC* tasks and minimizing the average slowdown for *BE* tasks.

C. Metrics

For *RC* tasks, we aim to maximize the *normalized aggregate value* (NAV) $\frac{aggregate\ value}{maximum\ aggregate\ value}$, where *aggregate value* is the actual aggregate value achieved for *RC* tasks and *maximum aggregate value* is the maximum possible aggregate value for *RC* tasks.

For *BE* tasks, we aim to maximize the *normalized average slowdown* (NAS) $\frac{SD_B}{SD_{B+R}}$, where SD_B is the average slowdown for *BE* tasks when *RC* tasks were treated as if they were *BE* tasks (no special treatment for *RC* tasks) and SD_{B+R} is the average slowdown for *BE* tasks when *RC* tasks were scheduled with the goal of maximizing their *aggregate value*.

D. Problem Formulation

We consider a stream of file transfer requests, each defined by a seven-tuple: <*source host*, *source file path*, *destination host*, *destination file path*, *file size*, *arrival time*, *value function*>. Requests arrive in an online fashion; that is, future transfer requests are not known a priori. Requests with a null *value function* are *BE* requests and those with a valid *value function* (like that discussed above) are *RC* requests. Hosts may have different capabilities (CPU, memory, disk speed, storage area network, network interfaces, WAN connection), and thus the maximum achievable end-to-end throughput may differ for each <*source host, destination host*> pair. External load at a source, destination, and intervening network may also vary over time, as may achievable transfer rates between a source and destination. Each host (source or destination) has a limit on the number of concurrent transfers that it can support. Our goal is to maximize the aggregate value obtained for *RC* tasks and minimize the average slowdown for *BE* transfers.

IV. Scheduling Algorithm

After presenting our formal problem statement and the metrics for the bi-objective problem, we now introduce the algorithm we have developed. We call the new algorithm *Response-critical (or RC) Enabled SEAL*: RESEAL.

A. Priority for Best-Effort Tasks

As noted above, RESEAL aims to minimize the average slowdown for *BE* transfers. In the implementation, we use the *expected slowdown* (also known as *expansion factor* or *xfactor*) of a task at any given time as the priority of the task. We define the *xfactor* for a file transfer task as follows,

$$\text{xfactor} = \frac{\text{Waittime} + TT_{load}}{TT_{ideal}}, \quad (5)$$

where TT_{load} is the estimated transfer time (TT) under the current load conditions and TT_{ideal} is the estimated transfer time under ideal (zero load and ideal concurrency) conditions. We describe below (Listing 2) how these values are calculated.

B. Priority for Response-Critical Tasks

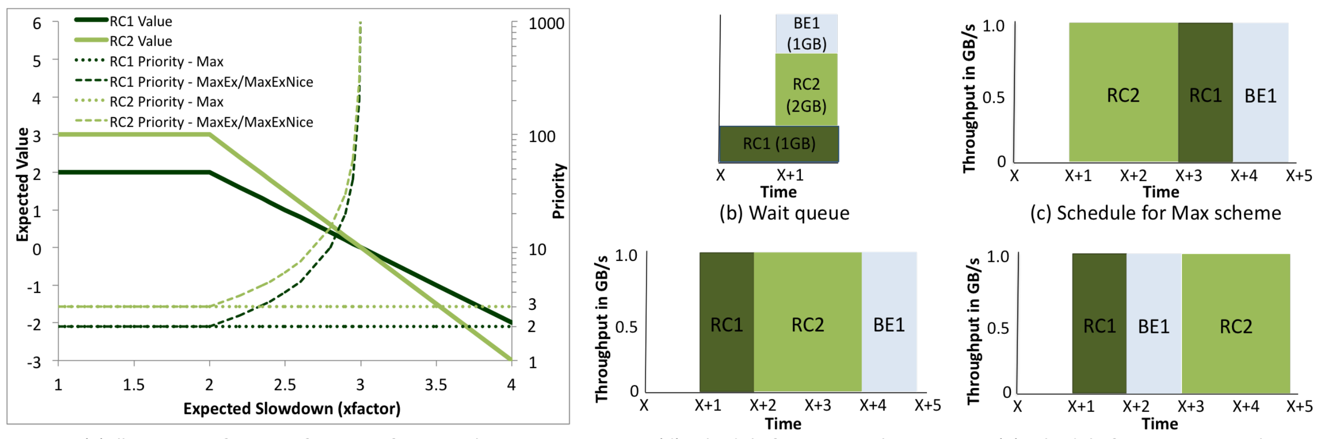
Since the goal for the *RC* tasks is to maximize their aggregate value, we prioritize them based on their value rather than their *xfactor*. We consider two approaches. The first approach uses the *MaxValue* of an *RC* task as its priority. The second approach prioritizes *RC* tasks based on both their importance and their urgency, which are measured by their *MaxValue* and current *expected value*, respectively. We define the current *expected value* in terms of its *xfactor* and *value* function,

$$\text{expected value} = \text{value(xfactor)}, \quad (6)$$

and the priority of an *RC* task as

$$\text{priority} = \text{MaxValue} \times \frac{\text{MaxValue}}{\max(\text{expected value}, 0.001)}, \quad (7)$$

where the quotient term specifies how far off the current expected value of a task is from its *MaxValue*. The higher



(a) Illustration of priority functions for RC Tasks
(b) Wait queue
(c) Schedule for Max scheme
(d) Schedule for MaxEx scheme
(e) Schedule for MaxExNice scheme

Fig. 3: Example illustrating RESEAL.

this ratio, the lower the expected value of the task compared with its *MaxValue* value and thus the higher its priority. We multiply this ratio by the *MaxValue* of the task in order to give higher priority to tasks with comparatively higher *MaxValue*s.

C. Prioritizing between BE and RC Tasks

Our algorithm needs to prioritize not only within the set of BE and RC tasks but also between them. For this purpose, again we consider two approaches. The first approach always gives higher priority to *RC* tasks over *BE* tasks. The waiting *RC* tasks are scheduled ahead of waiting *BE* tasks. If there is a waiting *RC* task, it would preempt as many running *BE* tasks as needed to enable the *RC* task to attain the same throughput it would achieve in the absence of any *BE* task in the system. We call this approach *Instant-RC*.

The second approach is as follows. Since the *RC* tasks yield *MaxValue* as long as they finish with a slowdown less than or equal to $Slowdown_{max}$, *RC* tasks do not need to be scheduled upon arrival. Ideally, the *RC* tasks can be scheduled such that they finish with a slowdown that is just under (and as close as possible to) $Slowdown_{max}$, if it can help minimize the other objective of minimizing the average slowdown of *BE* tasks. Thus, if the *xfactor* of an *RC* task is close (for example, >90%) to $Slowdown_{max}$, then the *RC* task is scheduled immediately with *dontPreempt* flag set. Otherwise, the *RC* task is given lower priority—it will be scheduled to run (without setting *dontPreempt* flag) if there is unused bandwidth after all high-priority *RC* tasks and *BE* tasks are scheduled. In addition to helping minimize the average slowdown of *BE* tasks, this approach can help increase the aggregate value for *RC* tasks. We call this approach *Delayed-RC*.

D. Three RESEAL Schemes

We define three schemes based on the ways in which priority can be defined for and between *RC* and *BE*.

Max uses the *MaxValue* as the priority for *RC* tasks, and the *Instant-RC* approach described above for scheduling *RC* and *BE* tasks.

MaxEx takes into account both the *MaxValue* and the current *expected value* for prioritizing *RC* tasks. Specifically, it uses Eqn. 7 to compute the priority for *RC* tasks. Like *Max*, it uses the *Instant-RC* approach for scheduling tasks.

MaxExNice, like *MaxEx*, uses Eqn. 7 to compute the priority for *RC* tasks. But it then uses the *Delayed-RC* approach of §IV-C to schedule *RC* and *BE* tasks. It thus minimizes the impact of *RC* tasks on *BE* tasks and other *RC* tasks. (*RC* tasks are nice to other tasks.)

All three schemes use *xfactor* to prioritize among *BE* tasks.

E. Detailed Example

We use the example in Fig. 3 to illustrate the differences among the three RESEAL schemes. The one source and one destination each have a maximum throughput of 1 GB/s (8Gbps). Fig. 3(b) shows the wait queue. An *RC* task RC2 (a 2 GB file) and a *BE* task BE1 (a 1 GB file) arrive at the time $t = x + 1$. Another *RC* task RC1 (a 1 GB file) has been waiting in the wait queue (it did not get scheduled because the source and destination were saturated with other *RC* tasks). At time $t = x + 1$, all other tasks complete, and only RC1, RC2, and BE1 need to be scheduled. Let the *xfactor* of RC1 be 2.35 at $t = x + 1$. The *xfactor* of both RC2 and BE1 is 1 at $t = x + 1$, since they both just arrived at $t = x + 1$. Let us assume that no more tasks arrive until $t = x + 5$.

Fig. 3(a) shows how the *expected value* changes as a function of *xfactor* for RC1 and RC2 as well as how the *priority* changes as a function of *xfactor* for the three schemes. Fig. 3(c) shows the schedule for the *Max* scheme. It schedules all *RC* tasks in the wait queue before considering *BE* tasks (unless the aggregate bandwidth for *RC* tasks is limited to a certain percentage of the maximum bandwidth at source and/or destination and that limit is reached: see §IV-F). *Max* prioritizes *RC* tasks based on their *MaxValue*. The *MaxValue* for RC1 and RC2 are 2 and 3, respectively, assuming A = 2 (see Eqn. 4). Hence, the algorithm schedules RC2 first, RC1 next and then BE1.

Fig. 3(d) shows the *MaxEx* schedule. Like *Max*, *MaxEx* schedules all *RC* tasks in the wait queue before considering *BE* tasks. But it incorporates the *expected value* of *RC* tasks in computing the *priority* for them (see Eqn. 7). As noted above, the *MaxValue* for RC1 and RC2 are 2 and 3, respectively. The *xfactor* for RC1 and RC2 are 2.35 and 1, respectively. Thus, we get an *expected value* of 1.3 and 3 for RC1 and RC2, respectively (see Fig. 3(a)). Substituting these values in Eqn. 7, the *priority* for RC1 and RC2 at $t = x + 1$ are 3.07 (2 $\times$ 2/1.3) and 3 (3 $\times$3/3), respectively. Thus, *MaxEx* schedules RC1 first, RC2 next, and then BE1.

Fig. 3(e) shows the *MaxExNice* schedule. *MaxExNice*, like *MaxEx*, uses Eqn. 7 to prioritize *RC* tasks, but schedules *BE* tasks ahead of *RC* tasks (even if the bandwidth limit for *RC* tasks is not reached) if an *RC* task's *xfactor* is not close to or greater than its $Slowdown_{max}$ (that until which the *RC* task retains its *MaxValue*). In this example, $Slowdown_{max}$ is assumed to be 2 and $Slowdown_0$ (that at which the *RC* task's value becomes zero) is assumed to be 3.

Listing 1 Scheduler, ScheduleHighPriorityRC, ScheduleBE, and ScheduleLowPriorityRC functions

```
function SCHEDULER($NT$)
    $W$.enqueue($NT$)
    Remove all completed tasks from $R$
    for task $\in$ $R, W$ do
        UpdatePriority(task)
    end for
    if !$W$.isEmpty() then
        ScheduleHighPriorityRC()
        ScheduleBE()
        ScheduleLowPriorityRC()
    else
        Up $cc$ for RC tasks if not sat and not $sat_{rc}$
        Up $cc$ for BE tasks if not sat
    end if
end function
function SCHEDULEHIGHPRIORITYRC( )
    $T$ = RC Tasks in $R \cup W$ with dontPreempt not set
    Sort $T$ in descending order of task.priority
    for task = $T$.peek() do
        continue if task.xfactor $\leq 0.9 \times Slowdown_{max}$
        if not $sat_{rc}$ then
            $R^+$ = Tasks in $R$ with dontPreempt set
            goalThr $\leftarrow$ FindThrCC(task, false)[1] $s.t.$ $R$=$R^+$
            Adjust goalThr to respect RC bandwidth limits
            Preempt task if task $\in R$
            $CL \leftarrow$ TasksToPreemptRC(task, goalThr)
            Preempt tasks in $CL$ and schedule task
            task.dontPreempt= true
        end if
    end for
end function
function SCHEDULEBE( )
    for task = $W$.peek() $s.t.$ task is BE do
        $CL_{src} = CL_{dst} = []$
        if not sat or isSmall(task) or task.dontPreempt then
            Schedule task
        else
            $CL_{src} \leftarrow$ TasksToPreemptBE(src, task)
            $CL_{dst} \leftarrow$ TasksToPreemptBE(dst, task)
            Preempt tasks in $CL_{src} \cup CL_{dst}$ and schedule task
        end if
    end for
end function
function SCHEDULELOWPRIORITYRC( )
    for task = $W$.peek() $s.t.$ task is RC do
        Schedule task if not sat and not $sat_{rc}$
    end for
end function
```

TABLE I: Summary of terms used.

Item	Description
NT	Set of new tasks
R	Priority queue of running tasks (ascending *xfactor*)
W	Priority queue of waiting tasks (descending *xfactor*)
TT_{load}	Transfer time under current load (based on model)
TT_{ideal}	Ideal transfer time (based on model)
TT_{trans}	Time the task has not been idle so far
xf_{thresh}	xfactor threshold to disable preemption for *BE* tasks
cc	Concurrency
$size$	Transfer size
$value()$	Value function of an *RC* task
β	User-defined variable for increasing concurrency
$maxCC$	Maximum concurrency allowed for a task
sat	Boolean - *true* means *src* or *dst* is saturated
sat_{rc}	Boolean - *true* means RC bandwidth limit reached for *src* or *dst*
throughput	Function to estimate throughput based on a model

Listing 2 UpdatePriority, ComputeXfactor, and FindThrCC functions

```
function UPDATEPRIORITY(task)
    if task is BE then
        task.priority$\leftarrow$task.xfactor$\leftarrow$ComputeXfactor(task)
        task.dontPreempt= true if task.xfactor> $xf_{thresh}$
    else if task is RC then
        $R'$ = Tasks in $R$ $s.t.$ task.dontPreempt is true
        task.xfactor$\leftarrow$ ComputeXfactor(task) $s.t.$ $R = R'$
        task.priority= $\frac{task.value(1) \times task.value(1)}{task.value(task.xfactor)}$
    end if
end function
function COMPUTEXFACTOR(task)
    [idealCC, idealThr] $\leftarrow$ FindThrCC(task, true)
    [bestCC, bestThr] $\leftarrow$ FindThrCC(task, false)
    $TT_{ideal} = \frac{task.num\_bytes\_total}{idealThr}$
    $TT_{load} = \frac{task.num\_bytes\_left}{bestThr} + task.TT_{trans}$
    return $\frac{task.WT + TT_{load}}{TT_{ideal}}$
end function
function FINDTHRCC(task, forIdealThr)
    thr=0; cc=0; dstload=srcload=0
    if ! forIdealThr then
        dstload = dst.cc;  srcload = src.cc
    end if
    do
        bestThr = thr; cc++
        thr$\leftarrow$ throughput(src, dst, cc, srcload, dstload, size)
    while (thr > bestThr $\times\beta$) and (cc < maxCC)
    return [cc, bestThr]
end function
```

MaxExNice thus schedules RC1 first since its *priority* is greater than that of RC2 and its *xfactor* is greater than its *Slowdown*$_{max}$, BE1 next at $t = x + 2$ as RC2's *xfactor* is still much less than its *Slowdown*$_{max}$, and then RC2. The *aggregate value* for RC1 and RC2 is 0.3, 4.3, and 4.3 for *Max*, *MaxEx*, and *MaxExNice*, respectively. The *slowdown* for BE1 is 4, 4, and 2 for *Max*, *MaxEx*, and *MaxExNice*, respectively. *MaxExNice* outperforms the other two schemes.

F. Formal Description

Listings 1 and 2 present the RESEAL algorithm. Table I describes the main data structures and terms.

Before continuing with the description of the algorithm, we explain two important issues: *concurrency* and *prediction models*. We use concurrency while transferring a file (in most cases) to improve performance. Specifically, to achieve concurrency at network, CPU and storage, our implementation exploits Globus GridFTP's support for partial file transfer (i.e., transfer *X* bytes of data from offset *Y*). We use this feature to perform multiple independent transfers, each of a partial file. We ensure that the partial transfer sizes are at least as big as the *bandwidth-delay product* of the given network link, in order to avoid additional overhead. Note that concurrency is related to, yet distinct from, *parallelism*—the latter implies using multiple TCP streams in parallel to accelerate transfers, and is used in Globus GridFTP [2] (and other tools such as BBCP [6]). Concurrency, unlike parallelism, exploits multiple CPUs more and helps with storage I/O performance as well.

One challenge is predicting how transfer performance improves as a function of concurrency. The `RESEAL` algorithm leverages a model from our previous work [28] to estimate transfer throughput. This model, trained offline with historical data, is used by the function *throughput* in Listing 2, line 73. It estimates throughput for a transfer given the desired concurrency level, known load (from ongoing transfers) at source and destination, and transfer size. It then applies a correction to account for current external (unknown) load, computed by comparing the historical data and the performance of recent transfers for the particular source-destination pair.

Now, we return to the description of the algorithm. The scheduling cycle repeats every n seconds; NT contains all new tasks that arrived in those n seconds. (In our implementation, $n = 0.5$.) At the start of each cycle, completed tasks are removed from the run queue R, new tasks are added to the wait queue W, and *xfactor* and *priority* values are updated for both *BE* and *RC* tasks. For *BE* tasks, the *xfactor* is computed by Eqn. 5, considering all tasks in R. For *RC* tasks, *xfactor* is computed by Eqn. 5, but considering only the preemption-protected tasks in R (as *RC* tasks can preempt all non-preemption-protected tasks in R if needed). Next, *priority* is computed for all tasks. For *BE* tasks it is the same as *xfactor*, whereas for *RC* tasks Eqn. 7 is used. In order to prevent starvation of *BE* tasks, preemption of *BE* tasks is disabled if their *xfactor* exceeds a certain threshold. (See Listing 1, lines 3–6 and the functions in Listing 2.)

The first step is to schedule the high-priority *RC* tasks in W (ones whose *xfactor* is greater than or close to $Slowdown_{max}$). If the aggregate throughput of *RC* tasks is less than the product of λ (a user-defined fraction, $0 \leq \lambda \leq 1$, to limit bandwidth for *RC* tasks) and the maximum achievable throughput for both source and destination (i.e., sat_{rc} is *false*), `RESEAL` schedules one or more high-priority *RC* tasks that are either waiting to run or running as low-priority *RC* tasks (the priority has since increased). These jobs are scheduled in descending order of their priority. The algorithm makes each *RC* task achieve a throughput that it would achieve if high-priority *RC* tasks in R and preemption-protected *BE* tasks (ones whose *xfactor* has exceeded a user-defined threshold) in R were the only tasks in the system, subject to the user-defined bandwidth limits (via λ described above) for *RC* tasks. We call this throughput the *goal throughput* for the *RC* task under consideration and compute this value using the model we referred to above. `RESEAL` preempts as many non-preemption-protected tasks as needed—the TasksToPreemptRC function identifies these tasks by computing the estimated throughput of the given RC tasks when non-preemption-protected running tasks are removed incrementally. The algorithm uses appropriate concurrency for the *RC* task (determined by using the model) to get a throughput as close to the goal throughput as possible. Preemption is also disabled for each scheduled high-priority *RC* task.

The algorithm next schedules the *BE* tasks in W in descending order of *xfactor*. If neither the task's source nor destination is saturated or if the task is small (<100 MB) or if the task's *xfactor* exceeds a threshold (*dontPreempt* flag is set), the algorithm schedules the task with appropriate concurrency (Function ScheduleBE in Listing 1, lines 35, 36). A task's concurrency is determined by using the FindThrCC function in Listing 2. If the waiting task's source and/or destination is saturated, then the algorithm considers preempting tasks associated with the waiting task's source and/or destination that are not preemption-protected. Among such tasks in R, it preempts one or more tasks whose *xfactor* is lower than that of the waiting task by a user-defined preemption factor *pf* (Listing 1, function ScheduleBE, lines 37–41). Function TasksToPreemptBE identifies candidate tasks to preempt in R. If the *xfactor* of a task in R is lower than that of the waiting task by *pf* or more, the running task is added to the candidate list (*CL*). The waiting task's *xfactor* is then recalculated but using a version of R that does not include the tasks in *CL*. If the new *xfactor* is sufficiently low, there are enough tasks in *CL*. Otherwise, this process is repeated until either *CL* is large enough or no more tasks are available for preemption.

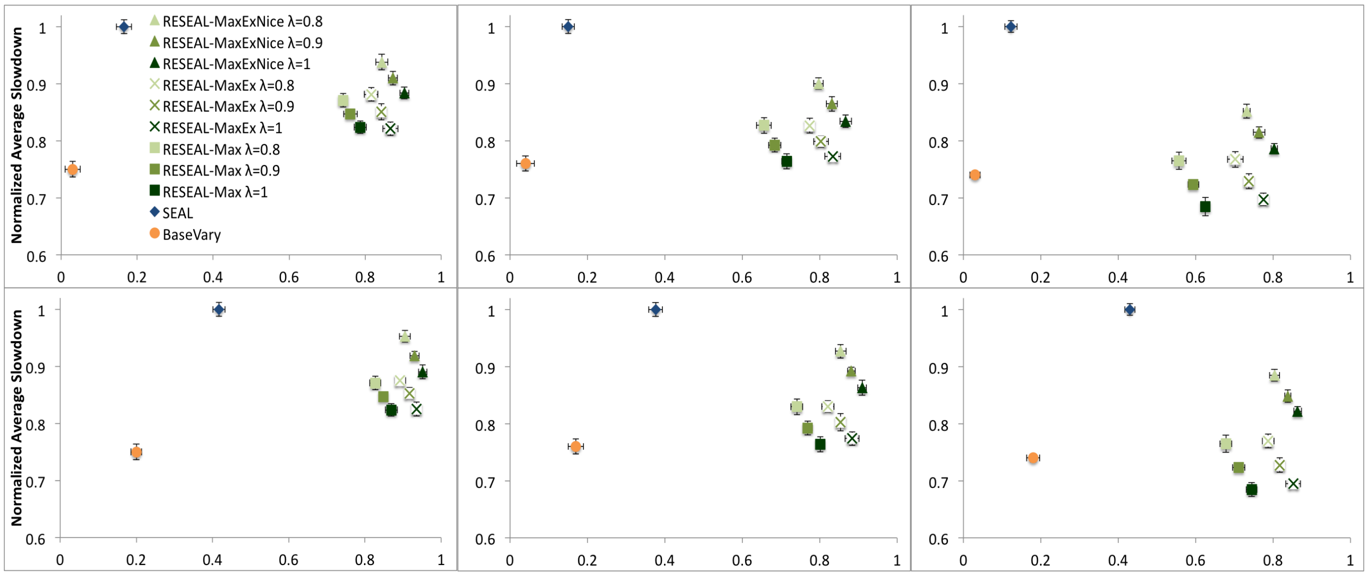
Next, `RESEAL` schedules the low-priority *RC* tasks in W. Each is scheduled only if the source and the destination of the tasks are not saturated and the aggregate throughput of the *RC* tasks associated with the source and destination is under the allowed bandwidth limit for *RC* tasks (Listing 1, function ScheduleLowPriorityRC). If there are no queued tasks for a scheduling cycle (i.e., W is empty), `RESEAL` ensures that any unused bandwidth, whether due to tasks completing or to reduction in external load, is used.

For this, we first consider the *RC* tasks in R in descending order of *priority*, and we increase their concurrency if their source and destination are not saturated and the aggregate throughput of the *RC* tasks associated with the source (and destination) is under the allowed bandwidth limit for *RC* tasks. Then, the algorithm considers the *BE* tasks in R in descending order of *priority* and increase their concurrency if their source and destination are not saturated (Listing 1, lines 11–14).

To determine whether the aggregate observed throughput of *RC* tasks exceeds $\lambda \times$ the maximum possible throughput at either the source or the destination (i.e., to determine whether sat_{rc} is *true* or *false*), we maintain a moving five-second average of observed throughput for each transfer. This moving average is also used for determining whether an endpoint is saturated or not. We conclude that an endpoint E is saturated if either of the following is true: (a) aggregate observed throughput for all transfers involving that endpoint is close ($\geq$95%) to the maximum possible throughput, as revealed by previous empirical measurements (or historical data); or (b) increased concurrency results in a proportionately insignificant increase in estimated throughput on three active links (less if fewer are active) involving that endpoint, meaning that if concurrency is increased by a factor F, throughput is increased only by a factor of $0.25 \times F$ or less. This is how we determine whether *sat* (in Table I and Listing 1) is *true* or *false*.

The pseudocode in Listings 1 and 2 is for `RESEAL-MaxExNice`. The pseudocode for `RESEAL-MaxEX` can be derived by removing line 20 in the function ScheduleHigh-PriorityRC and omitting the function ScheduleLowPriorityRC, which is no longer needed since all *RC* tasks that can be scheduled within the *RC* bandwidth limits will be scheduled in the function ScheduleHighPriorityRC itself. The pseudocode for `RESEAL-Max` scheme can derived by making the following changes on top of the changes mentioned above for `RESEAL-MaxEx`: In Listing 1, function UpdatePriority, replace line 54 with $R' = R$ and line 56 with *task.priority = task.value(1).*

Functions ScheduleBE, TasksToPreemptBE, ComputeXfactor, and FindThrCC form the `SEAL` algorithm. The rest of the



(a) `20%` RC tasks (b) `30%` RC tasks (c) `40%` RC tasks

Fig. 4: Results for `45%` trace. x-axis: NAV for *RC* tasks; y-axis: NAS for *BE* tasks.

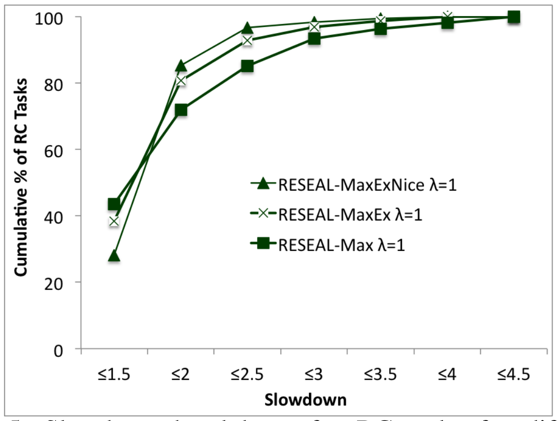


Fig. 5: Slowdown breakdown for RC tasks for different `RESEAL` schemes – `45%` trace.

pseudocode is the new addition for `RESEAL` algorithm.

V. Experimental Evaluation

We evaluated `RESEAL` with real traces in a wide-area environment using the Globus implementation [2] of the GridFTP [3] file transfer protocol. Although we evaluated our algorithms in the context of FTP, the scheduling problem that we consider is more general, and our algorithm is generally applicable for any data movement protocol.

Our goal is to evaluate the benefits of an algorithm that separates response-critical and best-effort transfers. To this end, we compare `RESEAL` with `SEAL` and a baseline algorithm `BaseVary` that varies concurrency based on file size. Although simple, `BaseVary` is a significant improvement over current practice in wide-area file transfers, where parallelism is exploited only on the network side for an individual file. We also examine whether, as we intend, `RESEAL` can meet response-critical transfer needs without a significant negative impact on best-effort transfers as total load is varied.

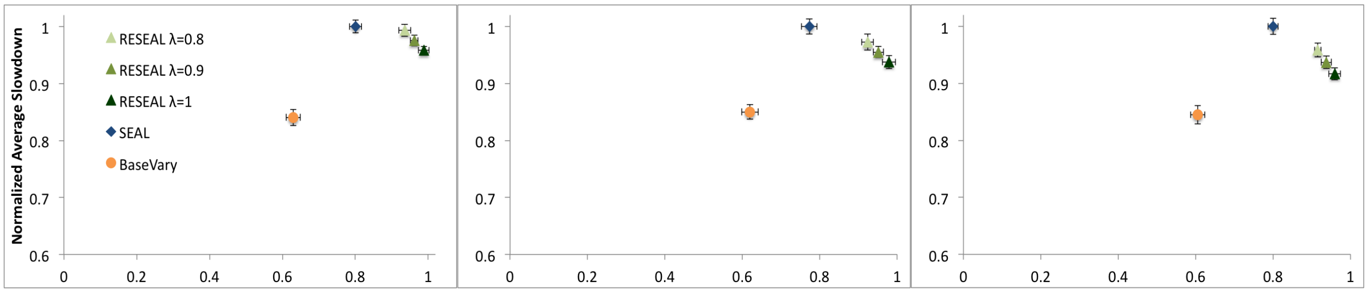
A. *Environment*

We conducted our experiments in a wide-area environment of six endpoints, each a data transfer node on a supercomputer: Stampede at the Texas Advanced Computing Center; Blacklight at the Pittsburgh Supercomputing Center; Darter at the University of Tennessee; Gordon at the San Diego Supercomputer Center; Mason at Indiana University; and Yellowstone at the National Center for Atmospheric Research. Each has a 10 Gbps WAN connection and is dedicated for wide-area data transfer. Stampede can achieve >9 Gbps aggregate disk-to-disk throughput; Yellowstone, Gordon, Blacklight, Mason, and Darter achieve ∼8 Gbps, 7 Gbps, 4 Gbps, 2.5 Gbps, and 2 Gbps, respectively. In our experiments, we used Stampede as a source and the other endpoints as destinations and ran at night and weekends to avoid disrupting other activities. Each result in §V-C is an average of at least five runs.

We note that requirements and usage reports from science communities [14], [43] suggest that large-scale data movement from experimental and observational facilities to a modest number of large compute facilities is a frequent requirement. Our experimental setup captures such a scenario.

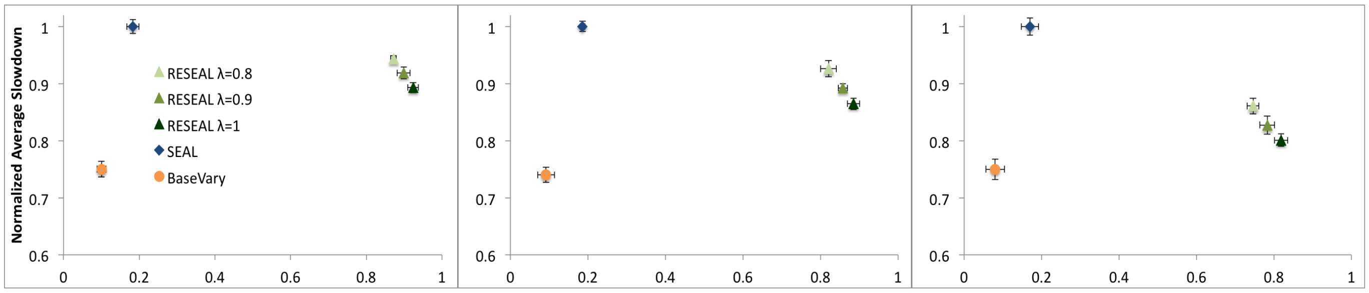
B. *Workload (Traces) Used for Evaluation*

We used real traces from Globus GridFTP servers (obtained from the Globus usage collector [20]) as workloads. These traces have transfer size and duration information but no identifiable information such as endpoint IP addresses. We picked the server that transferred the most bytes in a one-month period and the log corresponding to the busiest day (in terms of total bytes) in that period. Since our execution environment is a production infrastructure in continuous use, we were limited in the length of our experiments. Thus, we selected from the chosen 24-hour log several 15-minute traces with three different loads: the `25%`, `45%`, and `60%` traces, respectively. Average load of the 24-hour workload was ∼25%. We looked at all non-overlapping 15-minute windows in the 24-hour period and picked one with the same average load as the entire workload (25%). The coefficient of variation of 1-minute average concurrent transfers is approximately the same, too. We picked one that had the highest load (∼60%), and one with ∼45% load (which is in between 25% and 60%). We define *load* as the total volume of file transfers in the 15-minute trace divided by the maximum amount of data that the source can transfer in a 15-minute period. Since the maximum achievable



(a) `20%` RC tasks (b) `30%` RC tasks (c) `40%` RC tasks

Fig. 6: Results for `25%` trace. x-axis: NAV for *RC* tasks; y-axis: NAS for *BE* tasks.



(a) `20%` RC tasks (b) `30%` RC tasks (c) `40%` RC tasks

Fig. 7: Results for `60%` trace. x-axis: NAV for *RC* tasks; y-axis: NAS for *BE* tasks.

throughout for Stampede is 9.2 Gbps, the maximum that it can transfer in 15 minutes is ∼1 TB. Thus, the total transfer volumes in the `25%`, `45%`, and `60%` traces are ∼250 GB, 450 GB, and 600 GB, respectively.

For our experiments, we replay the transfers recorded in these traces. Since the traces do not include transfer endpoint identifiers, we distribute transfers randomly among the five destinations, weighted based on endpoint capacities. For each trace and for each destination, among the tasks that are ≥100 MB (all tasks <100 MB are scheduled on arrival), we picked X% of them randomly and designated them as *RC* tasks. We use X = 20, 30, 40 for our evaluation. We assigned a value function similar to the one shown in Fig. 2 to each *RC* task. We evaluated by assigning different value functions (by changing $Slowdown_0$={3,4} in Eqn. 3 and A={2,5} in Eqn. 4, $Slowdown_{max}$ was kept at 2). Because we have a large number of combinations across these parameters, we present only a subset of the results here. The trends were similar for others.

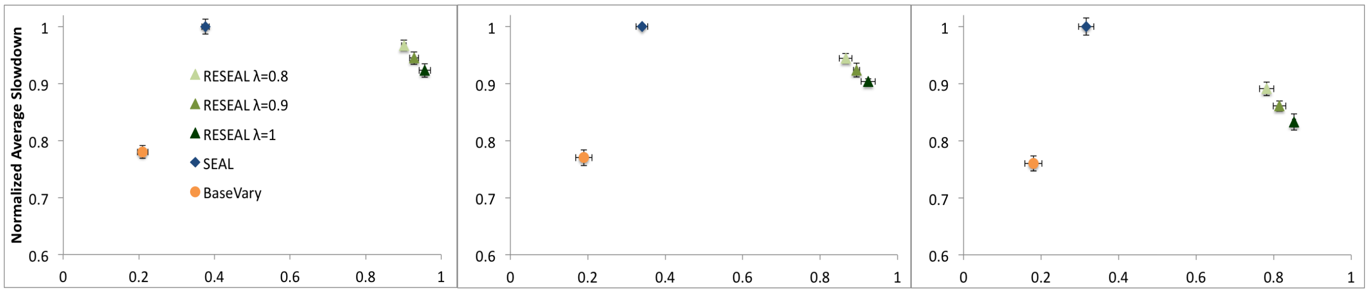
C. Evaluating Different RESEAL Schemes

We start our evaluation with the `45%` trace. We evaluate all three `RESEAL` schemes—`RESEAL-Max`, `RESEAL-MaxEx`, and `RESEAL-MaxExNice`—and use `SEAL` and a baseline scheme called `BaseVary` as two representative scheduling methods that do not differentiate based on transfer. In order to control the impact of *RC* tasks on *BE* tasks, `RESEAL` provides a hook for the system administrators to control the aggregate bandwidth that *RC* tasks can use at any time via the λ factor described in §IV-F. We evaluate all three schemes of `RESEAL` for λ = {0.8, 0.9, 1}, indicating that *RC* tasks can use up to 80%, 90%, and 100%, respectively, of the maximum available bandwidth at the endpoints. As noted in §III-C, we have a bi-objective problem—minimizing the average slowdown of *BE* transfers and maximizing the aggregate value for *RC* transfers. Both metrics introduced in §III-C, *normalized average slowdown* (NAS) and *normalized aggregate value* (NAV), take a value between 0 and 1, with a value close to 1 being desirable in each case. Also, in calculating NAS, the average slowdown for *BE* tasks, SD_B, is obtained by executing all tasks, including *RC* tasks as if they were *BE* tasks, under `SEAL`.

Fig. 4 shows the performance of nine variations of `RESEAL`: {`Max`, `MaxEx`, `MaxExNice`} $\times\ \lambda \in \{0.8, 0.9, 1\}$, plus `SEAL` and `BaseVary`. We executed each scheme for three different fractions of *RC* tasks and two different values of $Slowdown_0$, the slowdown value at which an *RC* task's value becomes zero. Fig. 4a shows the performance when 20% of the ≥100 MB tasks in the original `45%` trace are designated as *RC* tasks. The top and bottom graphs are for $Slowdown_0$=3 and $Slowdown_0$=4, respectively.

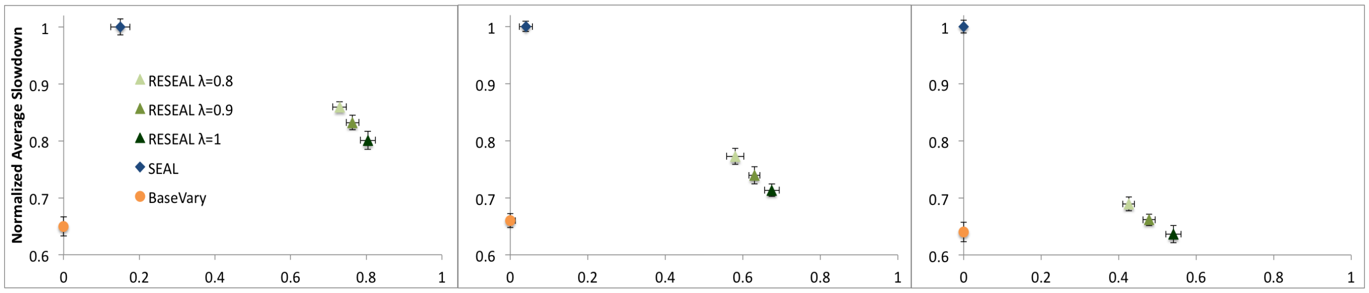
We see that all `RESEAL` schemes are far better than both `BaseVary` and `SEAL` in terms of NAV (x-axis), getting an aggregate value as much as 90% and 95% of the maximum aggregate value for $Slowdown_0$=3 and $Slowdown_0$=4, respectively. This is expected because `RESEAL` schemes differentiate *RC* and *BE* tasks, whereas `SEAL` and `BaseVary` schemes do not. The more significant observation is that we are able to meet the requirements of *RC* tasks, while having only a small impact on *BE* tasks. `RESEAL-MaxExNice` scheme achieves as high as 93% and 95% of the best NAS for $Slowdown_0$=3 and $Slowdown_0$=4, respectively, for the *BE* tasks. All `RESEAL` schemes do better than `BaseVary` for *BE* tasks even while providing significant value to *RC* tasks. The reason is that `BaseVary` assigns a static concurrency value for transfers without taking the current load information into account. `RESEAL-MaxExNice`, as in the example in §IV, is the best among the three `RESEAL` schemes. For example, `RESEAL-MaxExNice λ=0.9` achieves ∼87% of maximum aggregate value for *RC* tasks and ∼90% of maximum NAS.

The same trends continue when the percentage of *RC* tasks is increased to 30% and 40% of the ≥100 MB tasks in the trace (see Figs. 4b and 4c). Not surprisingly, both NAV and NAS decreases. The decrease in performance is more for `RESEAL-Max` (for both objectives) and `RESEAL-MaxEx` (for NAS) over `RESEAL-MaxExNice`. The fact that both `Max` and `MaxEx` prioritize all *RC* tasks over the *BE* tasks impacts the performance of *BE* tasks more as the percentage of *RC* tasks



(a) `20%` RC tasks (b) `30%` RC tasks (c) `40%` RC tasks

Fig. 8: Results for `45%-LV` trace. x-axis: NAV for *RC* tasks; y-axis: NAS for *BE* tasks.



(a) `20%` RC tasks (b) `30%` RC tasks (c) `40%` RC tasks

Fig. 9: Results for `60%-HV` trace. x-axis: NAV for *RC* tasks; y-axis: NAS for *BE* tasks.

increases. In addition, the fact that `Max` does not take the current value of the *RC* tasks into account for prioritization leads to a lower NAV with increasing percentage of *RC* tasks.

To further examine the tradeoff between the three schemes, we plot in Fig. 5 the cumulative percentage of *RC* tasks as a function of slowdown. We note that `MaxExNice` has the fewest *RC* tasks with slowdown $\leq$ 1.5, since it gives low priority to all *RC* tasks with an *xfactor* $\leq$ 1.8 (0.9 $\times$ $Slowdown_{max}$) and schedules them only when there is unused bandwidth after scheduling high-priority *RC* tasks (whose *xfactor* $>$ 0.9 $\times$ $Slowdown_{max}$) and the *BE* tasks. However, it has the most *RC* tasks with a slowdown $\leq$ 2 (and 2.5), as it gives the highest priority to *RC* tasks with an *xfactor* $>$ 1.8.

D. *Impact of Overall Load*

The preceding results were with a single trace (45% load). To assess performance as the total load on the network varies, we show in Figs. 6 and 7 results for `25%` and `60%` traces, respectively. Since `RESEAL-MaxExNice` clearly outperforms both `RESEAL-Max` and `RESEAL-MaxEx`, for the other traces, we present the results of only `RESEAL-MaxExNice`. We refer to `RESEAL-MaxExNice` as just `RESEAL` from now on. Also, since the results for *Slowdown*$_0$=4 are similar to those for *Slowdown*$_0$=3, we present results only for *Slowdown*$_0$=3 for the rest of the traces.

For the `25%` trace, the trends for `RESEAL` are similar to that for `45%`, with both NAS for *BE* tasks and NAV for *RC* tasks being higher than with `45%` for all three percentages of *RC* tasks (Figs. 6a, 6b, and 6c). The performance of both `SEAL` and `BaseVary` is much better for the `25%` trace than for the `45%` trace. The reason is that at low loads the average slowdown of all tasks is already low ($\sim$2.5 with `SEAL` and $\sim$2.8 with `BaseVary`).

Although `RESEAL` trends with `60%` are similar to those for `45%`, we noticed that both NAV for *RC* tasks and NAS for *BE* tasks for `60%` are better than those for `45%`. This result is counterintuitive: we expected the performance in terms of both objectives to be directly proportional to load. We did further analysis and experiments to explain this observation.

E. *Impact of Load Variation*

Visual examination of the two traces suggests that load (within the 15-minute duration) is relatively stable for the `60%` trace but varies considerably over time in the `45%` trace. To explore this phenomenon more rigorously, we define *load variation* in a trace as follows. Let $C_i(T)$, $i \in [0..duration(T)]$, be the average number of concurrent transfers during minute i of a trace T of length *duration*(T) minutes. Then the load variation of T, $\mathcal{V}(T)$ is the coefficient of variation of $\{C_i(T)\}$. We find that $\mathcal{V}$(`60%`) = 0.25 while $\mathcal{V}$(`45%`) = 0.51. A higher load variation likely means that there are more concurrent or overlapping *RC* tasks when load is high, which makes getting maximum or high value for all of them difficult. Since there will also be more *BE* tasks during high load periods, the negative impact of *RC* tasks on *BE* tasks will also be greater. Thus, we attribute the observed better performance on the `60%` trace to higher load variation in the `45%` trace.

To test this conclusion, we perform experiments with two more traces: 45% low variation (`45%-LV`) and 60% high variation (`60%-HV`). These traces are from the same GridFTP server as the others, but a different 24-hour period; $\mathcal{V}$(`45%-LV`) = 0.28 and $\mathcal{V}$(`60%-LV`) = 0.91.

Figs. 8 and 9 show results for `45%-LV` and `60%-HV`, respectively. `RESEAL` performs *better* on `45%-LV` than on `60%` (and `45%`), in terms of both metrics, and significantly *worse* on `60%-HV` than on `60%`. Note that though NAV for `BaseVary` is shown as zero in Figs. 9a, 9b, and 9c, the aggregate value in all three cases is negative.

Overall, RESEAL's efficacy depends on both total system load and load variation over time. Since current networks tend to be lightly loaded, we can consider the `25%` results, for which RESEAL meets *RC* needs easily with almost no impact on other transfers, as representative of the common case. With a 60% load (the highest observed in real traces), RESEAL is effective if the load variance is modest. We conclude that by using an appropriate scheduling method, response-critical transfers can be supported without resource reservations.

VI. Related work

Computational job scheduling (both parallel and distributed) has been studied extensively [38], [16], [33]. In contrast, the scheduling of file transfers among distributed resources has received less attention [10], [18], [32]. We are aware of only one effort that considered differentiated services [18], and that did not involve a bi-objective problem as here.

Several studies have used dedicated network channels to provide differentiated service for data transfers, both within [19], [44] and between [22], [40], [35], [42] sites. Others have investigated the co-reservation of heterogeneous resources, including storage and network resources [21], [18]. For reasons presented in §II-B, we pursue a reservationless approach.

Previous work on utility-function-based or user-centric approaches to scheduling [5], [12], [26], [34], has been performed in the context of computational jobs. In [29], we developed a file transfer scheduling algorithm that treats interactive file transfers and batch transfers separately, where batch transfers can tolerate significantly longer delay. However, that work did not consider *response-critical* tasks.

VII. Conclusions

We presented a new algorithm, `RESEAL`, to handle response-critical (*RC*) data transfers with minimal impact on other best-effort (*BE*) transfers. This algorithm uses user-supplied value functions for scheduling *RC* tasks. It treats *RC* and *BE* transfers differently, maximizing the aggregate value for *RC* tasks while minimizing the average slowdown for *BE* tasks. We evaluated `RESEAL` by using real traces and on a production system. We showed that the algorithm can achieve 90% of the maximum aggregate value for *RC* tasks with <10% impact on *BE* tasks even for logs with average load as high as 60%, when the percentage of *RC* tasks in the log is small, as long as the load variation is not high. The algorithm's performance decreases with increasing load, increasing load variation, and increasing percentage of *RC* tasks; increased load variation has the highest impact. Our results show that response-critical transfers can be conducted without resource reservations, exploiting the existing overprovisioning of wide-area networks, and through a carefully designed scheduling method.

Acknowledgments

This work was supported in part by the U.S. Department of Energy under contract number DEAC02-06CH11357 and RAMSES project, and the National Science Foundation, under grants ACI-1339757, ACI-1339798 and ACI-1440761.

References

[1] Advanced scientific computing research network requirements review, oct. 2012. https://www.es.net/assets/pubs_presos/ASCR-Net-Req-Review-2012-Final-Report.pdf.

[2] B. Allcock et al. Globus striped GridFTP framework and server. In *SC'2005*.

[3] W. Allcock. GridFTP: Protocol extensions to FTP for the grid, 2003.

[4] B. Allen et al. Software as a service for data scientists. *Commun. ACM*, 2012.

[5] A. AuYoung et al. Service contracts and aggregate utility functions. In *HPDC'2006*.

[6] BaBar Copy. http://slac.stanford.edu/~abh/bbcp/.

[7] Biological and Environmental Research network requirements. http://www.es.net/assets/pubs_presos/BER-Net-Req-Review-2012-Final-Report.pdf, 2012.

[8] Basic Energy Sciences Network Requirements Workshop, September 2010 – final report. http://www.es.net/assets/Uploads/BES-Net-Req-Workshop-2010-Final-Report.pdf.

[9] Bridges supercomputer. http://www.psc.edu/index.php/bridges.

[10] Z. Cai et al. IQ-Paths: Self-regulating data streams across network overlays. In *HPDC'2006*.

[11] A. Chervenak et al. Wide area data replication for scientific collaborations. In *GRID'2005*.

[12] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *CCGRID'2002*.

[13] E. Dart et al. The science DMZ: A network design pattern for data-intensive science. In *SC'13*, 2013.

[14] DOE data crosscutting requirements review, 2013. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/ASCR_DataCrosscutting2_8_28_13.pdf.

[15] R. Egeland, T. Wildish, and C.-H. Huang. PhEDEx data service. *Journal of Physics: Conference Series*, 219(6):062010, 2010.

[16] D. Feitelson et al. Parallel job scheduling – a status report. In *JSSPP'2004*.

[17] D. Feitelson et al. Theory and practice in parallel job scheduling. In *IPPS '1997*.

[18] I. Foster et al. End-to-end quality of service for high-end applications. *Computer Communications*, 27(14):1375–1388, 2004.

[19] B. Gibbard et al. Terapaths: End-to-end network path QoS configuration using cross-domain reservation negotiation. In *BROADNETS'2006*.

[20] GridFTP Usage Stats Collection. http://toolkit.globus.org/toolkit/docs/6.0/gridftp/admin/#gridftp-usage.

[21] J. Gu et al. StorNet: Co-scheduling of end-to-end bandwidth reservation on storage and network systems for high-performance data transfers. In *INFOCOM WKSHPS 2011*.

[22] C. Guok et al. Intra and interdomain circuit provisioning using the OSCARS reservation system. In *GridNets'2006*.

[23] High Energy and Nuclear Physics network requirements review. http://www.es.net/assets/Papers-and-Publications/HEP-NP-Net-Req-2013-Final-Report.pdf, 2013.

[24] High Performance SCP. http://www.psc.edu/index.php/hpn-ssh.

[25] Internet2 IP backbone capacity augment practice. http://www.internet2.edu/policies/ip-backbone-capacity-augment-practice/.

[26] D. E. Irwin et al. Balancing risk and reward in a market-based task service. In *HPDC'2004*.

[27] JetStream cloud system. http://pti.iu.edu/jetstream/index.php.

[28] R. Kettimuthu et al. Modeling and optimizing large-scale wide-area data transfers. In *CCGrid'2014*.

[29] R. Kettimuthu et al. An elegant sufficiency: Load-aware differentiated scheduling of data transfers. SC'2015.

[30] Y. Kim et al. LADS: Optimizing data transfers using layout-aware data scheduling. In *FAST'2015*.

[31] E. Kissel et al. Improving GridFTP performance using the Phoebus session layer. In *SC'2009*.

[32] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *ICDCS'2004*.

[33] J. Krallmann et al. On the design and evaluation of job scheduling algorithms. In *JSSPP'1999*.

[34] C. B. Lee and A. E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *HPDC'2007*.

[35] T. Lehman et al. Dragon: A framework for service provisioning in heterogeneous grid networks. *Comm. Mag.*, 44(3):84–90, Mar. 2006.

[36] Z. Liu et al. On causes of GridFTP transfer throughput variance. In *SC WKSHPS 2013*.

[37] Magellan: Cloud computing for science. http://www.alcf.anl.gov/magellan.

[38] A. Mu'alem et al. Utilization, predictability, workloads, user runtime estimates in scheduling IBM SP2 with backfilling. *IEEE TPDS*, 2001.

[39] L. Ramakrishnan et al. On-demand overlay networks for large scientific data transfers. In *CCGRID'2010*.

[40] N. Rao et al. UltraScience Net: network testbed for large-scale science applications. *IEEE Comm. Mag.*, 43(11):S12–S17, Nov 2005.

[41] V. T. Ravi et al. Valuepack: Value-based scheduling framework for CPU-GPU clusters. In *SC'2012*.

[42] J. Recio et al. Evolution of the user controlled lightpath provisioning system. In *Transparent Optical Networks, 2005*.

[43] XSEDE Report, 2013. http://ideals.illinois.edu/handle/2142/44872.

[44] J. Zurawski et al. The DYNES instrument: A description and overview. *Journal of Physics: Conference Series*, 396(4):042065, 2012.